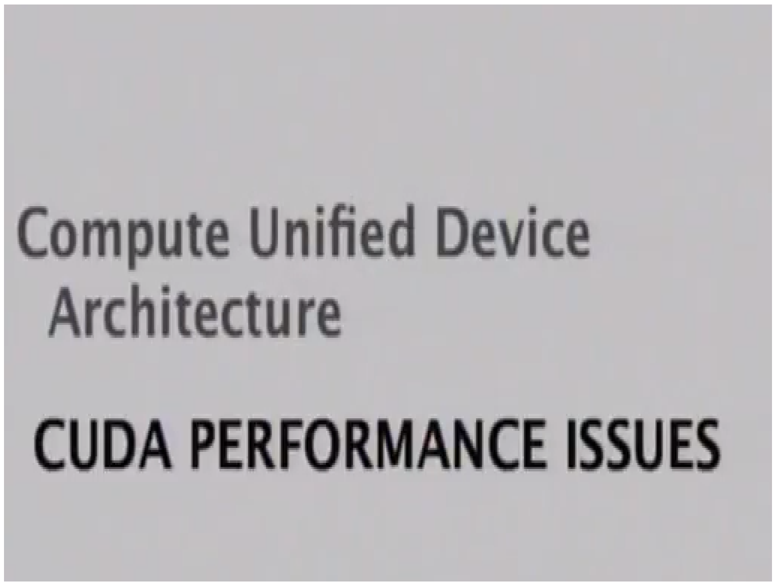
Parallel Computing

Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute of Technology – Delhi

Module No # 06
Lecture No # 26
CUDA (CONTD…)

(Refer Slide Time: 00:28)

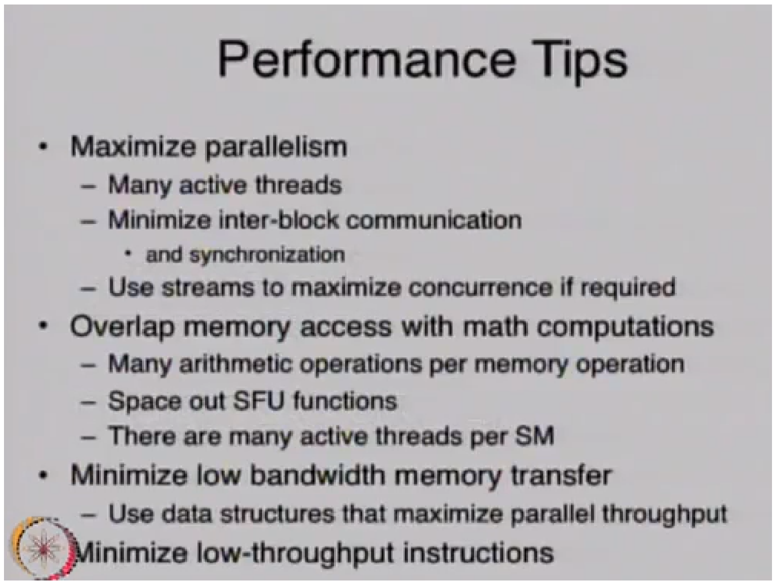


Compute Unified Device Architecture

CUDA PERFORMANCE ISSUES

We are going to talk about the performance issues that I had promised.

(Refer Slide Time: 00:33)



Performance Tips

- Maximize parallelism
  - Many active threads
  - Minimize inter-block communication
    - and synchronization
  - Use streams to maximize concurrence if required
- Overlap memory access with math computations
  - Many arithmetic operations per memory operation
  - Space out SFU functions
  - There are many active threads per SM
- Minimize low bandwidth memory transfer
  - Use data structures that maximize parallel throughput
- Minimize low-throughput instructions

So essentially what are the things to keep in mind you are trying to program in CUDA, In fact many of these things are not really CUDA specific although some of them are things that you would want to keep in mind when you are writing parallel programs okay of course you are want to maximize parallelism and from CUDA point of view that means as many threads okay. It also means as many threads in the context of block non block breakup right we know that every synchronization within a block we can handle it reasonably efficient.

Anytime you require synchronization across the block either there is some busy wait somewhere which says the block done right or you are going to start a new company. And so when you designing your kernels try to see if all communications or most of the communications can be structured within the block okay and the anytime there is this communication there is got to be some kind of synchronization to ensure that whatever you are sending is actually reaching there.

And whenever you can use streams you should that is again the issue of maximizing parallelism they are not necessarily the only way to maximize parallelism but there are one way again stream are running different kernels. So they are not taking in the same grid not added would make them difference even if you are in same grid but not in the same block with similar issue to deal with ahh and so now you have to think about what should go in a block.

Maybe I want the entire thing to be one block because blocks are easy to synchronize it is possible right one thread just do it does more and then block size becomes your size of parallelism the amount of parallelism that you get yet is the number of thread that you put in a block one problem is that one block as to run in one SM which means other SM’s are ideal you may say I have got say 16, thirties odd SM’s I can run 16 kernels in parallel.

So I will run 16 different things assuming as I have 16 different things to do or 16 different SM’s would you want to do that why not that is one issue but it is the same amount of work right. So you may be run out of memory on shared space but on the GPU it does not matter whether all the threads are organized so you have organized your parallelism into 1024 threads right and they are only they are reading up all the memory.

So you are saying that different kernels would be requiring their own blocks of memory that is potentially one issue suppose that is not an issue. As it stands today only one context can be run at one time so these different kernels run on different SM's have to come from one context right one application cannot be from different applications that may happen but that require much more tight interaction between the OS and the device which means NVIDIA and Microsoft have to get or other people have to not much if they merge that can happen.

But they have to talk very much more closely than they do that is possible they do not have to merge to have happen as a matter of fact they have been trying to get that feature in their at least 6, 7 years now it has not happen because they are not the same company. So if they were the same could have happen by now but anyway that since that is not there what are the other ahh problems.

Even if you allow multiple contacts suppose there is only our application you wanted to parallelize that is how you think even in the CPU domain I have got a large amount of computation to do and let us run it in parallel not worrying about one user running email application on one of the course. So the problem would be that there are only fixed number of threads you can run for a block so 1024 which means fixed number of warps right 32 wraps which mean if one of these 32 wraps is not ready to run in a given clock then you cannot run.

So total number of wraps that you have is limited and you would want many more wraps at least when you are talking about the system wide number of wraps meaning as the total device wide number of wraps you would want significantly more than 32 wraps to be run even on one SM probably you may want more than 32 warps now all right. So the amount of parallelism that you would get I am making one block do everything is going to go down or even a small number of blocks to everything small number being size of the number of SM's in your system.

For example okay you have to think further by not limiting yourself for saying I have got KSM's I am going to run K blocks each block is going to do K of the wrap. Well that number is not going to remain the same for every kernel but something so as a rule of thumb something between 4 and 16 K is probably a good somewhere in that range is highly likely that whatever your application is going to suits all that many number of blocks and then each block does one over K of 1 over 4 K whatever 10 K of the work.

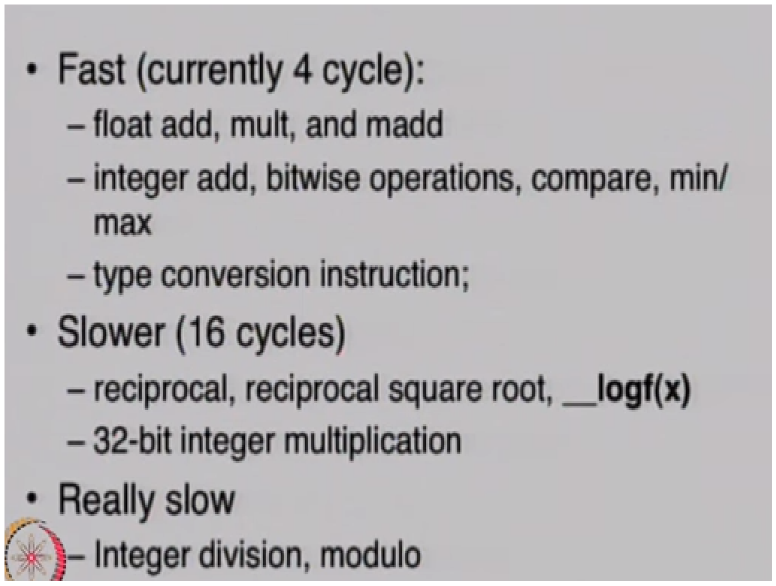
So maximizing parallelism is one part of the story okay now within the code there we have looked at the architecture right there is a arithmetic logic unit ALU stuff there are special function unit called SFU there is fewer of those you have to run many run a what worth of instruction across spread across multiple clock cycles for a SFU instructions and then their memory instructions.

And typically you would see read from memory do something on that then write the result into that memory that would be a natural sequence of instructions you will find often in any typical program what does that mean? I read something and just to give you the sense or to put actual numbers here in fact I think I have few numbers in an upcoming slide in fact let us go there we will talk about this different between different type of instructions and ho their distribution matter let me go through the last two items which we will talk about a little later also.

The other would be when you do read memory you know that you are doing parallel reads right you are not saying read the variable X we are actually saying that threads reads variable X that thread reads variable X that thread read variable X and 16 of them reading variable X their own version of variable X. So do these reads together fetch memory data items right like one capsule of data in an efficient way what should these pattern be so that you can do this in efficient way.

That is the third thing to think about and then minimize instruction that take long to run that is probably should not even be fourth item there but sometimes you can take short cuts there are expensive ways to do certain things the inexpensive ways to do certain things the inexpensive way to do the same thing probably with some error and you might take make that choice some times.

(Refer Slide Time: 11:29)



- Fast (currently 4 cycle):
  - float add, mult, and madd
  - integer add, bitwise operations, compare, min/ max
  - type conversion instruction;
- Slower (16 cycles)
  - reciprocal, reciprocal square root, **__logf(x)**
  - 32-bit integer multiplication
- Really slow
  - Integer division, modulo

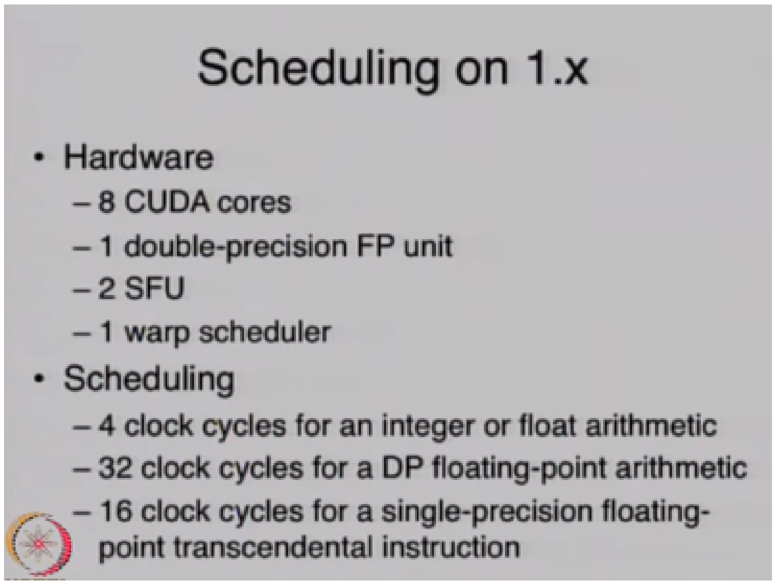
This is for 1 point X okay the SFU for example can take reciprocal can do a square root can do log and a it takes 16 cycles to do that most instruction will take four cycles at X to Y multiply X with Y compare minimum whatever but integer division takes significant mode. If you know that then you would get rid of integer division right.

In fact floating point division is much faster than integer division so you might replace their integer if you can in certain situations replace it with floating point of division floating point to an integer what you mean? What is the range of floats? Min float and max float that’s much bigger than – 2 to the 32 – 1 to 2 to the 32 – 1 right of – 2 to the 32 to 2 to the 32 – 1. Obviously not all the integers will be represented you could not just do floating point location for integer division but in certain situations you can.

For example if you know that integer is floating point representable then you might do it so these are the number I was talking about. This are pipe line okay so every time yu can schedule half a work it will scheduler things it is done for half work is done even if there is only 8 as it the slide shows only 8 CUDA code 16 wraps 16 threads have been scheduled okay. Internally this hardware is pipeline right so in the given clock it is going to run 8 of them next clock it is going to run another 8 of them.

And so they becoming after each other four cycles later you are going to get the first five cycle you are going to get the setup.

(Refer Slide Time: 14:35)



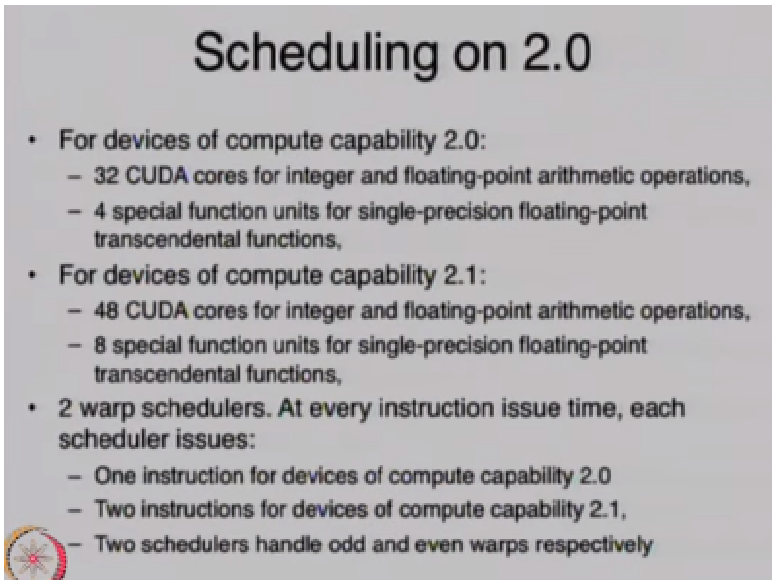
Scheduling on 1.x

- Hardware
  - 8 CUDA cores
  - 1 double-precision FP unit
  - 2 SFU
  - 1 warp scheduler
- Scheduling
  - 4 clock cycles for an integer or float arithmetic
  - 32 clock cycles for a DP floating-point arithmetic
  - 16 clock cycles for a single-precision floating-point transcendental instruction

So scheduling is something that will take 4 clock cycles for most similar instructions 32 clock cycles it has come significantly I think it is now four clock cycles for a four or 8 clock cycles for double precision floating point and 2 dot X architecture and 16 clock cycles for some function the once that go in SFU. And on 2.0 there are 32 CUDA codes but they are divided into groups of 16 so 2 groups of 16 and does a matter of fact 2 dot in fact it says here.

2.1 it is 3 groups of 16 so 3 it says 42, 48 CUDA codes per SM there more special function units in the most recent set of hardware and there are two wraps are scheduled in the same clock which means that since there are 16 half work is scheduled at the time so that 16, 16, 16 so you can schedule one kernel this one wrap here and one other kernals another wrap here or one kernals one block one warps here and the same kernel is another block and another warps here.

(Refer Slide Time: 16:16)


Scheduling on 2.0

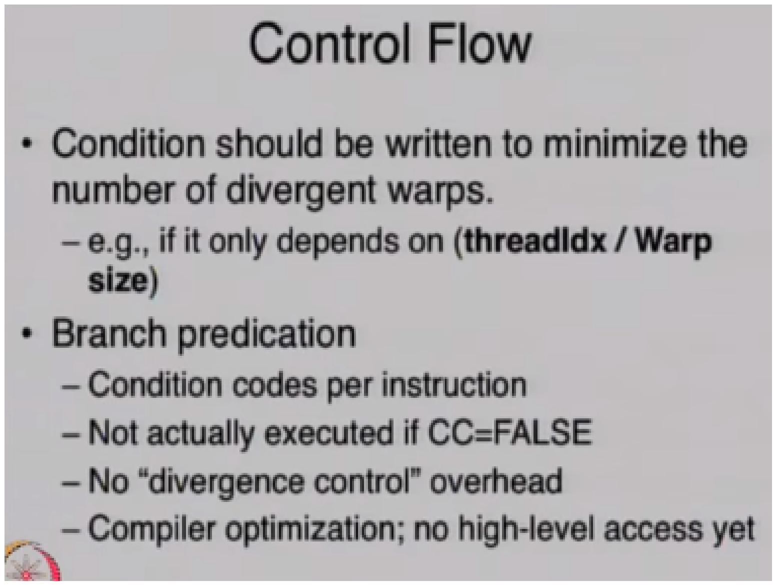
- For devices of compute capability 2.0:
  - 32 CUDA cores for integer and floating-point arithmetic operations,
  - 4 special function units for single-precision floating-point transcendental functions,
- For devices of compute capability 2.1:
  - 48 CUDA cores for integer and floating-point arithmetic operations,
  - 8 special function units for single-precision floating-point transcendental functions,
- 2 warp schedulers. At every instruction issue time, each scheduler issues:
  - One instruction for devices of compute capability 2.0
  - Two instructions for devices of compute capability 2.1,
  - Two schedulers handle odd and even warps respectively

And the rate at which you can schedule them is also quite much faster and so that the last statement is probably less of an issue for the most recent set of devices but the first three but you can sink about how to increase your parallelism how to have memories mixed with non-memory with computer instructions and what kind of addressing pattern would be good for a given instruction.

So that all the of the memory capsule comes in efficiently and so we those three things we are still going to talk about the other thing that is again something that is reasonably bad for even the newer ones especially bad for the older architecture was branch okay which means that if your branching with some threats in a warp trying to do X and the other threads of the warp trying to do Y then there is significant overhead for it.

One is that those two are X and Y serialized but the other is keeping track of which threads are to run when all the if and else and if and else that may be nested you require a significance amount of overhead. Now I said that now you also get this branch predication where instead of thinking of it has if then if then do it else otherwise do that you think of it has do this and this every instruction with a few of them disabled okay.

(Refer Slide Time: 18:27)



Control Flow

- Condition should be written to minimize the number of divergent warps.
  - e.g., if it only depends on (**threadIdx / Warp size**)
- Branch predication
  - Condition codes per instruction
  - Not actually executed if CC=FALSE
  - No "divergence control" overhead
  - Compiler optimization; no high-level access yet

It is not good for nesting or at least deep nesting bit it does not need to do any branch cracking like who has diversion which are them converging together again this is something that you do have direct control over this is part of the PTS instruction and the language has no way this if condition to be predicates although it might make sense for the language it had a micro where you can direct the instruction should be predicated.

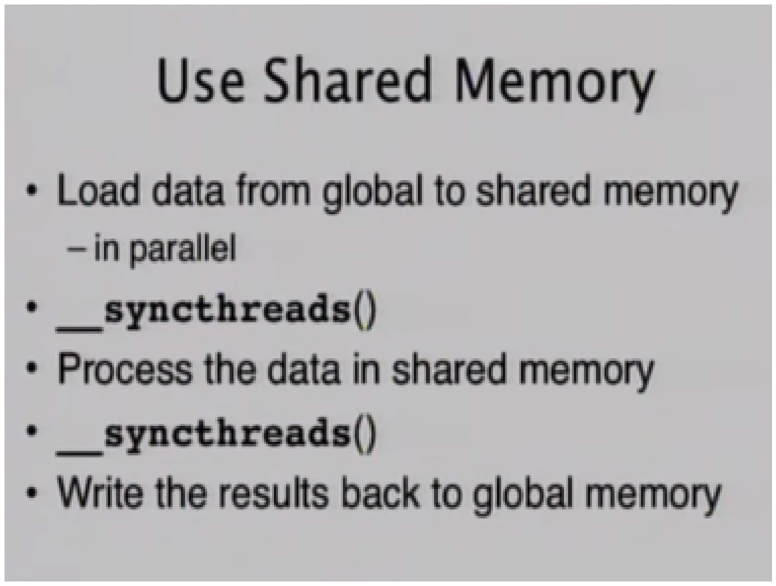
Currently the compiler has some heuristics if there are small sets of instructions within the if and else the few things that you are doing then it is going to use predicates and once it is get complicated enough for it to use predicates its falls back to the more general branch tracking okay. So message on this slide if you can avoid if do so okay. It is okay to do if within a warp in the sense that if all the thread of warp are true you can pay no price for it.

If all the threads of a warps or false you pay no price for it sometimes you can organize your competition such that for example sometimes you want to do something for add things something for the even things. If you do the natural way you say 0 thread number 0 is handling item number one is handling item number one then every warp is going to have half warp and half even things.

If instead you say if you are from 0 to block size divided by 2 then you are doing with even things block size 2 to the block size you are doing the odd things. Assuming no boundary conditions which you can arrange properly either the entire warp will be handling even data item or the odd day so that is not a condition that in practical sense. Sometimes it would be also useful to instead of saying if something do this you take the condition code and multiply by result right.

For example if you want to produce in the result in the given variable if the condition is true then you can say simple take the condition result multiplied but the variable and put it in the result assuming that otherwise you are going to put 0 okay. So the things of that sort typically would be good idea to do when conditional branching or conditions are inexpensive or expensive.

(Refer Slide Time: 21:46)

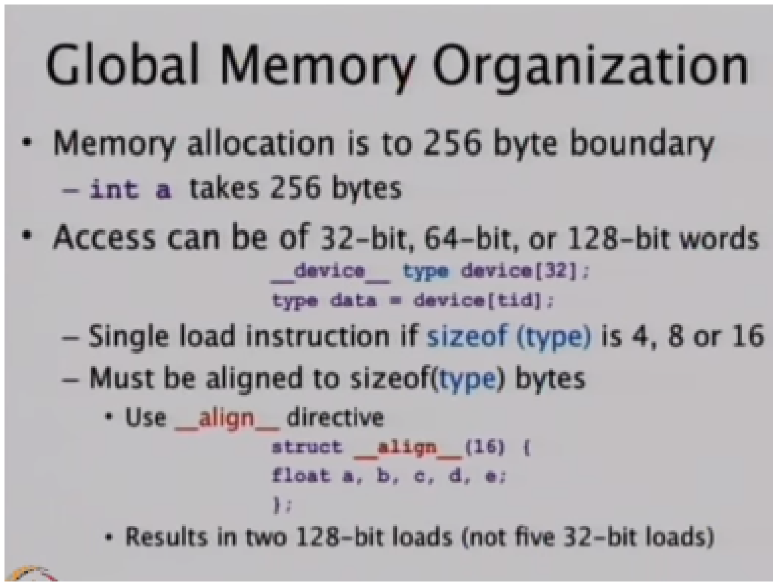


Use Shared Memory

- Load data from global to shared memory
  - in parallel
- **`__syncthreads`**()
- Process the data in shared memory
- **`__syncthreads`**()
- Write the results back to global memory

Use shared memory we already talked a lot of about that whenever you can you share memory whenever you can do not use sync threads. For example if you want to keep your syncing or miscommunication as much as possible within your warp then you do not need to use sync threads. It is essentially typically different blocks will be a different stage right but if you want everybody to come to the same stage then whoever as passed that stage becomes not a candidate for scheduling anymore okay.

(Refer Slide Time: 22:38)



Global Memory Organization

- Memory allocation is to 256 byte boundary
  - `int a` takes 256 bytes
- Access can be of 32-bit, 64-bit, or 128-bit words

```
__device__ type device[32];
type data = device[tid];
```

  - Single load instruction if sizeof (type) is 4, 8 or 16
  - Must be aligned to sizeof(type) bytes
    - Use __align__ directive

```
struct __align__(16) {
float a, b, c, d, e;
};
```

    - Results in two 128-bit loads (not five 32-bit loads)

So that reduces your working sets size that is all okay now let us talk about the organization shared memory we have already discussed these things are in banks of memory and the address organization is that contiguous address go to different banks so bank 0 will have the address 0 in it like 1 will have the address in it banks 15 will have the address 15 in it and 16 will come back here okay.

And think of this memory has a 16 ports okay so what that means is that if you are reading from different banks you can read it in parallel if you are reading from the same bank then you have to wait for that data to come to the port okay. And so from the shared memory point of view if all the different threads on a wall are reading from different banks you get excellent performance if they are reading from the same bank then they will be serialization.

So that parallel I/O will become serial right it will take as many clock cycles as the number of thing coming out of with that okay. And that number that I was looking for actually we have not seen I thought I have it somewhere the difference in the number of clock cycles it takes for data to come from memory versus the number of clock cycle to execute the instructions we have seen that right.

Four clock cycle is an example of number of cyclist to execute an instructions the number of cycles to get data from memory is 200 in that is the difference I am not saying exactly the 200 but it is something to the order of 40, 50 times. As a result of which you can do a lot of memory but this is pipeline it does not mean that you say read something from memory and now for 100 clock cycles you cannot read anything or whatever the 200 clock cycles you cannot read anything you can issue your reads at the rate you can launch instructions.

It is just that given reads data item will come back 100, 200 clocks cycles which means and there is no branch prediction or advance scheduling on this device that means is if after that read you are going to use that data item you have taken that what out of equation for 200 clock cycles right. Because until that thing comes you cannot use if it instead you were to organize your computation in such a way that you a read few things in several different variables local variables and then you do the computation on that.

What that will do is because you are reading independent memory location into independent shared memory of register they can also see there is no dependence among these instructions at some point you are going to go and use that variable which in which you read something long back and hopefully by that time either the data has arrived or it is about to arrive. So this does not get out commission for very long period okay.

So that then you will wait for 190 cycles not very much if you can only shoot then reads right remember that you will almost in fact guarantee to never be able to issue ten reads in ten blocks because you do not have monopoly over the hardware their hopefully other warps waiting. So your subsequent reads will probably read about 20 to 25 clock cycles and so if you make ten reads it has taken you about 250 clock cycles to make them okay.

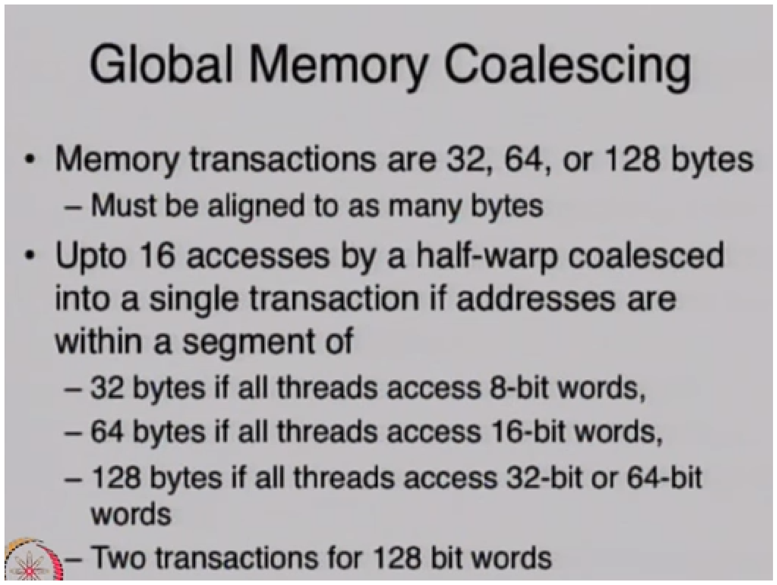
As long as you have enough warps run okay the memory organization the global memory you read data in bulks 256 byte is for example an atom size and you read in a typical instruction four byte am going to read an integer from memory I am going to read a float from it. Sometimes you can read bigger things you can read a double but overall you are doing 16 saturates. So if you can ensure that your 16 floats which amount is 64 bytes come in one atom of 256 bytes you can get it together and get it in single memory instructions into memory you say 256 lights from the address.

In older cards 256 was not the only atom cells can read it in 64, 128 or 256 bytes that is the maximum magnum cells. So you can there are three different read instructions all right and you say I want to read 64 bytes from the address that is what the SM will say and it will say that on behalf of all the threats. So if the threats entire access comes within one of those atom then you get it now because the atom size is 256 bytes the cache line is also 256 bytes which is somewhat large cache line.

For example a typical cache line on CPU would be 64 pipes okay and so which means false sharing is something to be especially careful about. The other thing that you want to make sure is the addresses are aligned right so it is not that you can read only 256 bytes it has definitely has to be contiguous it also has to be aligned at 256 byte boundary okay. And so it is good idea for your struts also to be aligned right so typically you say read the 1oth element of my strut.

For example in this example you are saying strut align 16 which means align it in 16 bytes float ABCD and so this entire strut will being at 16 byte boundary or 16 X whatever X may be X being an integer and you can read the entire strut in one short when you copy a strut to an another strut you can entire strut in one shot.

(Refer Slide Time: 31:29)



Global Memory Coalescing

- Memory transactions are 32, 64, or 128 bytes
  - Must be aligned to as many bytes
- Upto 16 accesses by a half-warp coalesced into a single transaction if addresses are within a segment of
  - 32 bytes if all threads access 8-bit words,
  - 64 bytes if all threads access 16-bit words,
  - 128 bytes if all threads access 32-bit or 64-bit words
  - Two transactions for 128 bit words

So ultimately your coalescing of memory will have to depend on getting continuous chunks in a single read instruction single write instruction okay which means you organize you organize your data structure such that when many things are ready the reading contiguously let me stop here. Okay let us continue we were talking about the performance issue and particular we were talking about the memory side of N’s.

And there are two kinds of memory accesses shared memory and the global memory and there is different kind of behavior that you want some time those to behaviors conflict meaning that you want to sometime bring global data memory to shared memory right. So you copied something from local memory write something to shared memory and if you follow the same pattern of accessing so usually you would say I have got an array and index number I1, I2, I3, I4 of the indexes that the different threads of the given warp or for that matter a given block.

Compute and then they read I1, I2, I3, I4 from the global memory of write at I1, I2, I3, I4 in the shared memory and that read pattern and this write pattern for you the thread is the same that would be the natural thing to do. For example if you are reading contiguous things then you would say I am reading 1 you are reading 2 you are reading 3 you are reading 4 and then I am writing 0 into 0 X location you are writing 1 into location and so on.

And in some cases it may be fine to do that pattern for the global memory but not for shared memory whereas it may be fine you are making the shared memory faster but not the global memory okay and so then you have to think how to make sure that it is a good pattern for both okay. And good which means you have to understand what is the good pattern for each one of them and roughly briefly but we have to talk about is that you want contiguous chunks to be read from global memory.

So all the threads produce and address and those address or collated into one or more depending on how those addresses look in terms of contiguity and hence so forth into one or two memory access it remains me I had said that the memory atom size is 256 megabytes which is true for allocation. You allocate in atoms of 256 megabytes 256 bytes and the allocation addresses also aligned 256 bytes.

But you can read at most 128 bytes from memory in one shot okay so that atom size is 128 bytes you can read less than that if you want. So if you only need 32bytes you only read 64 you will read 64 you only need 128 you will read 128. And if you need more then you will read in you will send multiple memory transactions to the memory. So let me understand you are asking I want 512 bytes of data to come from local memory to shared memory right.

It depends on how you want to read it right if those 512 bytes of data are in one contiguous location and global memory right then you know you can read it in 4 different transactions that is up to you. So the hardware says there are two parts of the story one is think of it has the bunch of processors which is for us the bunch of threads producing a bunch of addresses and with each address there is a certain size I may want to read one byte at given address I may want to read hint I may want to read a double.

And so how much can I read in one instruction is one part of it one side of the story okay and I suggest here you can read the 32 bits or 64 bits or 128 bits in a single instructions okay now there are up to 32 things making either 32 bit or 64 bit or 128 bit request. In 1.X you are guarantee that it will be half of that because half of warp running and then in fact a warp was run over 4 cycles because there is only 8 of them and it is grouped in groups of half warp because 16 request can go to the memory at one shot or 16 request can be collated into one or more request.

In 2 dot X that is not necessary the case because 2 half warps can be scheduled at the same time and so it may be all 32 but so ignoring those kind of details there is so warp of read to happen right and either its half warp or full warp whatever each instruction is meaning one instructions the entire warp as one instruction that instruction says read 32 bits or it may say read 128 bits.

You can actually read a vector of XYZW you can see where this coming from it is all started with graphics where things groups always go in groups 3 or 4 and that is why you can read 4 things in one shot.

There is one instructions in CPU you will have typically one instructions to read one 32 bit here you can read one instruction to read 4 32 bits of data so it is a vector read. One vector read per lane right there are 32 lanes and so 32 people are saying one vector for me one vector for me one vector for me and one vector for me so that there are 32 vectors to be run. So now when you think of that single instructions overall effect then we are reading 32 times 4 bytes right.

And now you the question is how much can be read in one shot from the memory that is how much threads want to read and think of now having kind of go between where threads says I want to read and go between says you want this I will take your orders I will combine those orders and I will send the minimum number of orders necessary to the memory.

If everybody is reading contiguous blocks then I can say you want 32 bits so total there are 16 people wanting 32 bits right so how many bytes 64 bites to be read and am going to send one 64 byte request to the memory okay. Similarly if entire 32 bit wide warp to read one inch right then it is 32 times 4 bytes and so I can send all of that in one memory request as long as those 128 bytes that are being read in the combined warp or 128 byte align and of course the size is 128 bytes.

So here we so far discussing from the global memory prospective okay some of the discussion remains the same when you are doing shared memory you are saying all of these want to read something and for a given warp there are so many orders from the global memory you collate those orders and then you decide to place those orders with the memory. So first of all there are banks even in the global memory side but they are not that transparent.

So they are not made visible to the programming model banks in the shared memory is actually made visible to the programming model that is one okay. Second the more important thing is the banking is related to how you address them I said I want to read address 0 the next says address one next side wants to read address 2 and the 16 threads wants to read address is 0 to 50. So when I combine those orders I will say I want to read 16 bytes starting at 0.

Now if the banks are visible to me then I will say where does the address 0 reside besides in bank number 0 where does address one resides I say bank number 1 and address resides in bank number 15. So if I want to say I want to read 16 things 16 hints starting at address number 0 what does the amount to it amounts to the reading address 0 of each bank right. So as long as address are straight through the different banks right you go 0, 1, 2, 3, 4 whatever the rank size is then start again.

Then contiguous address will come from different banks as long as they aligned it is definitely a single memory instruction. From program point of view this is single instructions where everybody is saying read given at address it is a single read instruction they are not different address to read but that is how the SIMD works. And from this waiter point of view who is taking every body order and feeding it to memory.

It is still a single memory instruction because it is saying read 16 bytes starting at 0 the address are so assigned that they are coming from different times. So in this example what you are seeing is the top example we have some type and we have an array of certain type and you can say data = device in given array locations MITID. And this again we as the slide says talking about global memory organization that will amount to a single memory instruction.

Different TID will be accessing different addresses but everybody is saying read something at the given offset from the device. And as in this case because you are using directly TID those offsets are contiguous because all the TIP where contiguous. Now depends on the size of type the size of type is 4, 8 or 16 we get 32, 64 or 128 bit instructions. The threads ID's within the warp are contiguous you do not directly see the ID's of SP's.

There are launched all the SP's it is possible that some of the threads are not actually executing this instructions right they diverged there is an if condition another is predicate that has disabled that specific lane but processor them you never really think in terms of processor ID it is a thread ID and you know that for a given warp thread ID will be contiguous it would go from 0 to 16 warp size is 32 0 to 31.

But it will go from some 32 X to 32 X + 1 – 1 or X + 1 in parenthesis then your blocks becomes really small. No because you can actually have a much more friendly relationship with other warps of your blocks then you do with warps in the same kernels but not in your block right. So you can do more synchronization so you can share more data share the shared memory all of that will go away because it is done it is very small size okay.

So now looking at the other example where it is strut which is been aligned to 16 bytes meaning that it is going to take aligned 16 byte address and you say X = Y and that X is what 5 floats it can be by the compiler. It will be compile into 2 instructions ADC and D will fit in one instruction so it is going to read ADC and D in one instruction and E in the other instructions and copy it from X to Y.

So instead of making 5 float loads it is going to make on vector loads of float 4 and then other float 32 bit load okay. So coalescing is happening at that level and as we discussed memory transaction will be one of the sizes 32, 64 128 bytes and if all the instruction addresses fit in one of these good it is going to take a single instructions. If they do not then multiple instructions may be required for example if the first address is 0 and the second address is 5 and the third address is take multiples of 5 right what is going to happen?

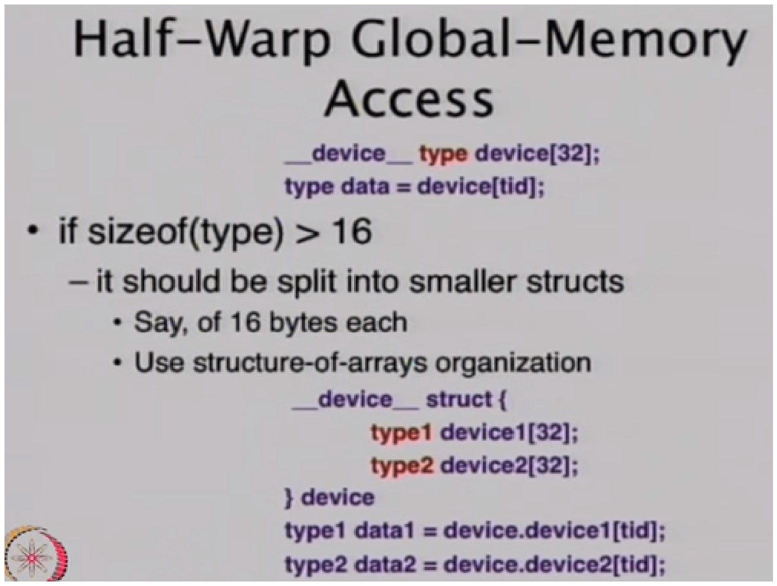
The global memory read so you are reading 0, 5, 15, 20 and so on because the waiter is not going to able to put them all in one plate it is going to send them all in their own individual place right. It is going to combine the things it can it is going to leave the rest as their own instructions and now a days meaning with 2 dot X any time it can combine it does in specially in 1.0 1.1 time frame there are lot more restrictions in that they had to be uniformly spread and they all have to be aligned.

Now two of them might be reading the same address there was some unnecessary restrictions that have since been removed so it would behave in as expected way as long as your sending addresses can be coalesced. There is nothing like everything is running on that SP right because it is done at an instruction by instruction basis when one instruction is run and it has to come from half of warp right.

It is only that instructions addresses can be coalesced because anything else comes in next clock cycle whenever that is going to be scheduled and in fact I did not the I was looking for the number about how fast things run and I have put in a slide in the end when we are going to review these things about how fast the different instructions run so there is coalescing algorithm which are not going to go through.

It basically whatever we talked about what it implements it looks for address can be combine together and it combines them and there is even in 2 dot X there are certain situations in which we are distribution of address will not get combined for example ahh some later lane reading the previous address previously lane reading the previous and that address and some other lane in the middle reading something far away okay.

(Refer Slide Time: 51:34)



Half-Warp Global-Memory Access

```
__device__ type device[32];
type data = device[tid];
```

- if sizeof(type) > 16
  - it should be split into smaller structs
    - Say, of 16 bytes each
    - Use structure-of-arrays organization

```
 __device__ struct {
          type1 device1[32];
          type2 device2[32];
} device
type1 data1 = device.device1[tid];
type2 data2 = device.device2[tid];
```

So this same set of rules also being demonstrated in this where you are reading two different locations in this strut organize. You first leave device one and this is stage in global memory and then you read device two you could instead simply read device what is the difference and this is again a more concrete example that I was hand writing about at the end of last class.

Blindly look at strut or arrays that is correct but one step deeper why not strut of arrays are not going to work. So look at the first instructions right everybody is reading something right they are reading contiguous chunks if you have read whole thing would that still happy? Why? What happens when it read it together? The meaning is that I say device strut device block = device = device TID where you would have to declare it also of that TID right.

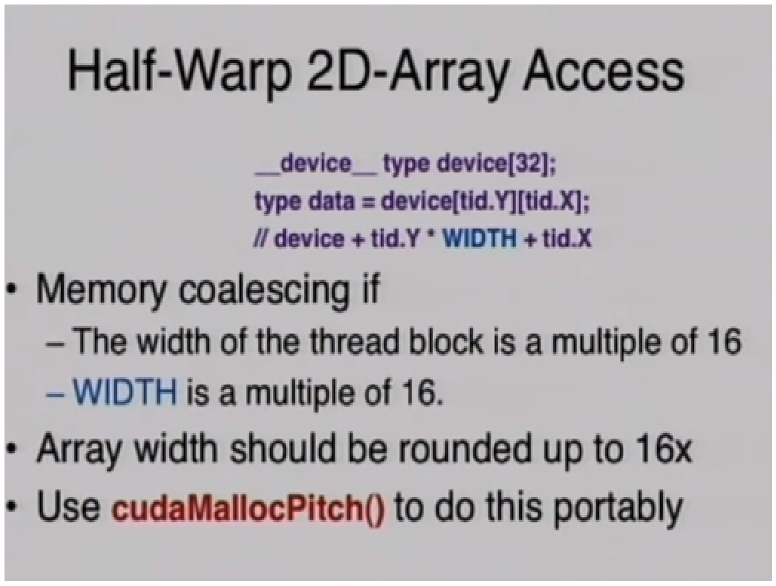
Declare as type one as type once device type one device one type 2 device 2 and the array would be outside. Device of size 32 but if I take the entire device right can it be how would it be the whole device the array is outside right. But will that be one instruction depends on type 1 type 2 right.

Suppose type 1 type 2 let us put some names there hint it will coalesce because they are all going to read the entire array so they are all going to have the same addresses why you would do that is not clear some reason everybody needs the same data then everybody is sending he same address to be read and how many have to read 32 times 2 integers. I am assuming that you can optimize the compiler I do not want to deal with the what compiler what actually does.

We will talk about what it can do what it might be able to do so array assignment can be done in one shot strut assignment can be done in one shot. So small static array can be done in one shot easily it does not have to open a loop for it so everybody is reading the same array everybody is producing the same address and the address is starting somewhere wherever the device 0 begins 32 times 2.

Not TID everybody is reading entire because you have just said type data another array is you reading something into local array from the global array. Yeah I was saying if you read the whole array it is the same thing you reading 0^{th} position you reading first position you reading Nth as long as the compiler recognizes okay. And it will be coalesce and contiguous and it of course it would not fit in one instructions it can be done in two right.

(Refer Slide Time: 57:28)



Half-Warp 2D-Array Access

```
__device__ type device[32];
type data = device[tid.Y][tid.X];
// device + tid.Y * WIDTH + tid.X
```

- Memory coalescing if
  - The width of the thread block is a multiple of 16
  - WIDTH is a multiple of 16.
- Array width should be rounded up to 16x
- Use cudaMallocPitch() to do this portably

There is also if you making 2D arrays you want to make sure that your dimensions are multiples of 16 or you use something called pitch malloc meaning that it is going to pad the each row some number of whites so the next row beings at proper multiple depends on the type proper means multiple of the atom size the type size within this okay. If they were small then it would be perfectly fine because it is fit in one instructions okay.

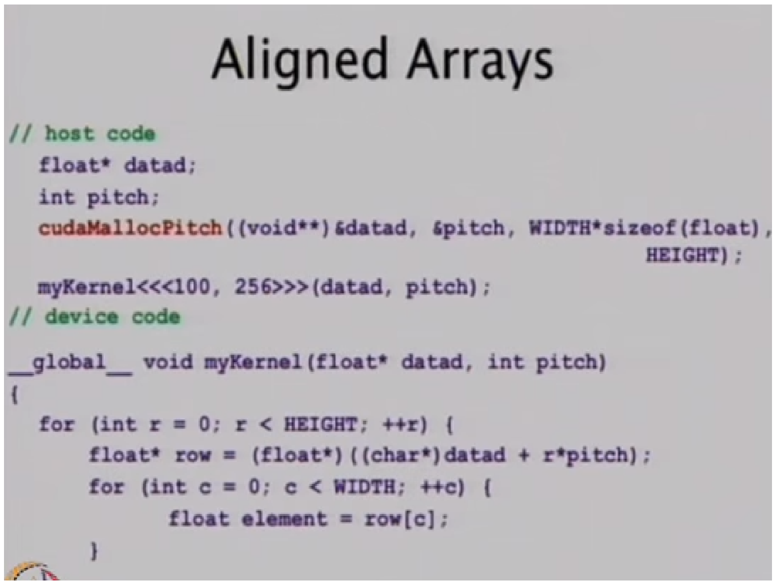
So this is again I am not going to dwell on this it is something that we have to keep in mind if you are going to use 2D array either your width the dimensions of the array are going to multiple of the size here it is says 16 right. So basically every half work will be aligned to 15 times whatever the size of the data is right and so the alignment will be in data terms some multiple of that data aligned with 16 times that which is exactly where we started with right.

We said that if it reading blocks of 128 bytes and it is aligned by 128 bytes which comes from 16 times the read size of each thread or if it is blocks of 64 reading for the entire block and 64 comes from 16 times the read elements size of each thread right. So as long as your width is 16 of whatever type you are using it is going to get align each of the reads as long as contiguous ID's contiguous indexes into this array they are going to get aligned.

And so there is this notion of CUDA malloc pitch instead of CUDA where you tell it what is that alignment what is that size that I am going to align so it takes X as well as Y dimension that then

I gives you data that is expanded so that the alignment that can be taken care of and then you have to also access it in certain fashion. Let malloc which is simply a malloc with a slightly bigger size align size so that if you are not using the entire width you can leave a little bit of buffer on the right hand side.

(Refer Slide Time: 1:01:01)



Aligned Arrays

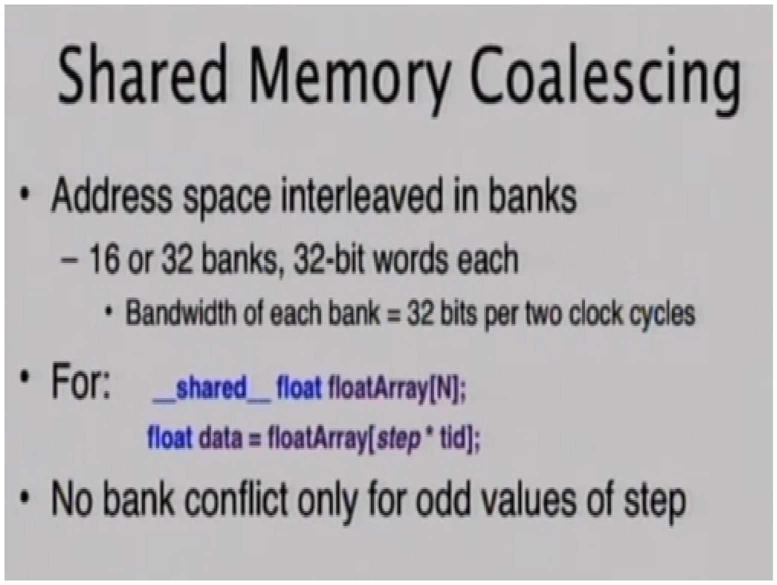
```
// host code
  float* datad;
  int pitch;
  cudaMallocPitch((void**)&datad, &pitch, WIDTH*sizeof(float),
                                                      HEIGHT);
  myKernel<<<100, 256>>>(datad, pitch);
// device code

__global__ void myKernel(float* datad, int pitch)
{
  for (int r = 0; r < HEIGHT; ++r) {
      float* row = (float*)((char*)datad + r*pitch);
      for (int c = 0; c < WIDTH; ++c) {
             float element = row[c];
      }
```

And there are ways of reading data from mallc pitch in a standard way okay. So in this case you are saying I am going to read at multiples of this pitch okay and you provide that pitch to the reader which is going to access it so pitch is something that is being returned by CODA malloc pitch and in the reader you depending on how many rows have crossed you are saying you need to offset your address by that many pitch multiples because every time you go to next row.

There is pitch worth of data that you are that is blank and there is buffer so you have to skip across that.

(Refer Slide Time: 01:02:00)



Shared Memory Coalescing

- Address space interleaved in banks
  - 16 or 32 banks, 32-bit words each
    - Bandwidth of each bank = 32 bits per two clock cycles
- For:
```
__shared__ float floatArray[N];
float data = floatArray[step * tid];
```
- No bank conflict only for odd values of step

On the shared memory side it is still contiguous that you like addresses should be contiguous what about the differences I was saying that shared memory and global memory have how does that make a difference? Why when would a contiguous read from so you are saying I need to bring some block of data from global memory to shared memory and as a thread am going to read this block and I am going to write that block into the same memory.

When would it not work well ensuring that it is contiguous on both sides when would not work well? Meaning one of them is not getting accessed in coalesce way think about it we know the structure of shared memory or I am asking you the question. I am simply taking a block of data from global memory and saving it local memory where is an example where both of them are contiguous and I am taking contiguous block form global memory and saving it in contiguous in shared memory.

Where is example? where I am not accessing the memory in coalesced fashion and am going to stop.